(2)

	GE	Form Approved OMB No. 0704-0188

1. AGENCY USE	January 1991	3. REPORT TYPE AND DATES COVERED memorandum

4. TITLE AND SUBTITLE Experiments with Dataflow on a General-Purpose Parallel Computer	5. FUNDING NUMBERS N00014-85-K-0124 N00014-88-K-0738 N00014-87-K-0825 MIP-8657531
6. AUTHOR(S) Ellen Spertus and William J. Dally	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139	8. PERFORMING ORGANIZATION REPORT NUMBER AIM 1272
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Information Systems Arlington, Virginia 22217	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES None	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution of this document is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The MIT J-Machine [2], a massively-parallel computer, is an experiment in providing general-purpose mechanisms for communication, synchronization, and naming that will support a wide variety of parallel models of comptuation. Having universal mechanisms allows the separation of issues of language design and machine organization [4]. We have developed two experimental dataflow programming systems for the J-Machine. For the first system, we adapted Papadopoulos' explicit token store [12] to implement static and then dynamic dataflow. Each node in a dataflow graph is expanded into a sequence of code, each of which is scheduled individually at runtime. For a later system, we made use of Iannucci's hybrid execution model [10] to combine sev-

(continued on back)

14. SUBJECT TERMS compilation hybrid architectures parallelization MIMD dataflow			15. NUMBER OF PAGES 18 16. PRICE CODE $3.00
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNCLASSIFIED

eral dataflow graph nodes into a single sequence, decreasing scheduling overhead. By combining the strengths of the two systems, it is possible to produce a system with competitive performance. We have demonstrated the feasibility of efficiently executing dataflow programs on a general-purpose parallel computer.

Keywords: compilation, parallelization, dataflow, MIMD, hybrid architectures.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1272 January 1991

Experiments with Dataflow on a General-Purpose Parallel Computer

Ellen Spertus and William J. Dally

Abstract

The MIT J-Machine [2], a massively-parallel computer, is an experiment in providing general-purpose mechanisms for communication, synchronization, and naming that will support a wide variety of parallel models of comptuation. Having universal mechanisms allows the separation of issues of language design and machine organization [4]. We have developed two experimental dataflow programming systems *for the J-Machine. For the first system,* we adapted Papadopoulos' explicit token store [12] to implement static and then dynamic dataflow. Each node in a dataflow graph is expanded into a sequence of code, each of which is scheduled individually at runtime. For a later system, we made use of Iannucci's hybrid execution model [10] to combine several dataflow graph nodes into a single sequence, decreasing scheduling overhead. By combining the strengths of the two systems, it is possible to produce a system with competitive performance. We have demonstrated the feasibility of efficiently executing dataflow programs on a general-purpose parallel computer.

Keywords: compilation, parallelization, dataflow, MIMD, hybrid architectures.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124. The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation and IBM Corporation, and by a National Science Foundation Graduate Fellowship.

1 Introduction

The dataflow programming model is attractive because it exposes parallelism in computer programs, a crucial step in compiling for multiprocessing computers. Once a program has been converted into a dataflow graph, only necessary constraints among the operations remain, and the program can, ideally, be spread across many processors. In the past, this has been done by special-purpose dataflow machines such as DDM1 [5], the Manchester Dataflow Machine [7], Monsoon [13], and Sigma-1 [15] that directly execute dataflow graphs. In addition to research in pure dataflow architectures, there is a growing interest in developing hybrid architectures, such as the EM-4 [14], that take advantage of the parallelism found by dataflow methods without sacrificing the straight-line efficiency of von Neumann machines [6]. Our approach is to develop a system to execute dataflow programs on the J-Machine, a massively-parallel general-purpose computer [2].

The J-Machine was not specifically designed to support dataflow execution but instead to provide universal mechanisms for concurrency, synchronization, and naming that support many parallel programming models. Having universal mechanisms allows the separation of programming model issues from issues of machine organization [4]. Our task was to find ways to utilize the J-Machine's mechanisms to meet the requirements of dataflow programs. Additionally, we sought to find the best possible representation of dataflow programs to match the J-Machine.

Specifically, our first system involved translating each node of a dataflow graph into a sequence of code, where execution proceeded sequentially within each sequence, but the order of the sequences was determined at run-time. The second approach, motivated by a desire to lessen run-time scheduling overhead, made use of Iannucci's work on hybrid architectures [10] and Traub's work on dataflow graph sequentialization [21] to produce a single sequence of code for several dataflow graph nodes, allowing some scheduling to be done at compile-time. Both systems used dataflow graphs produced from the dataflow language Id [11]. This paper describes our experience and results with these systems.

1.1 Background

1.1.1 Id

Id is a mostly-functional language originally designed for programming dataflow computers [11]. Interesting features include I-structures and mechanisms for loop parallelization. I-structures are data structures that bypass the inefficiency of array modifications in purely-functional languages, where an array must be copied every time it is modified. Copying is not necessary for I-structures. Slots are initialized to *empty*, and read requests of an empty cell are silently deferred until the data is available. Writing a cell more than once denotes a run-time error.

While I-structures prevent Id from being purely functional, the language remains deterministic, an arguably more important property. The unbounded latency of an I-structure read is one of the reasons fine-grained scheduling is needed, in order to use the time efficiently that would otherwise be wasted waiting for a read request to complete.

A major source of parallelism in Id programs comes from loops. The semantics of Id is such that instructions from different iterations of a loop can be executed out of order as long as data dependencies are obeyed. More details about loop statements will appear later in the paper.

1.1.2 The J-Machine

The J-Machine [2] is a massively-parallel MIMD computer based on the Message-Driven Processor (MDP) [3], a custom chip. For this research, we used a simulator of a 32-node J-Machine [9]. Each processor has 260K (4K on chip) of 32-bit words augmented with 4-bit tags. Tag types include booleans, integers, symbols, pointers, and cfutures. A cfuture is used for synchronization to represent an empty location and typically is written into a memory location before the actual data value is ready. If an attempt is made to operate on the cfuture, a fault occurs.

The MDPs communicate with each other by sending messages through a low-latency network. When a message arrives at its destination, it is placed on the message queue, and a new task is created when the message reaches the head of the queue.

1.2 Overview

In the next section, we describe a straightforward method of implementing Id on the J-Machine, based on Papadopoulos' explicit token store (ETS) [12]. In the following section, we describe the system we built to simulate Iannucci's hybrid architecture on the J-Machine, focusing on the run-time data structures used to support the style of synchronization used on his hybrid architecture, and on loop parallelization. In the conclusion, we discuss the strengths and weaknesses of the two systems and describe our plans to combine them into an efficient implementation of Id.

2 ETS on the J-Machine

2.1 The Explicit Token Store

In a dataflow graph representation of a program, nodes represent operators, and arcs represent dependencies. *Tokens* are the mechanism for carrying data values on these arcs. Abstract dataflow machines have a waiting-matching unit that matches tokens destined for dyadic (two-input) operators with their partners. A token consists of several components:

1. A value to be operated on.
2. A *context*, indicating what instantiation of the graph it belongs to. (This will be more fully explained in the section on dynamic dataflow.)
3. The *destination address*, corresponding to the node to which it should be delivered.
4. A port number, indicating whether it is the "left", "right", or sole input.

Left and right tokens with the same context and destination address must be matched with each other and sent to their destination address together to be executed. It is usually more efficient to *explicitly* store tokens that arrive before their partners than to implement a waiting-matching unit directly [12, pp. 44–45]. In an ETS strategy, tokens that arrive before their partners are stored in ordinary memory. When a left token, for example, is processed, the appropriate memory location is checked for its partner. The memory address is a function of the destination address and context, both of which the left and right tokens share. If that location is not empty, it must contain the right token, and the operation is performed. If the location is empty, the left token is stored there, to be retrieved when the right token arrives and checks that location.

2.2 Static Implementation

Our first experiments with dataflow on the J-Machine involved *static* dataflow, in which dataflow graphs must be acyclic and nonreentrant. This eliminates the need for contexts, because the static discipline ensures that only one instantiation of a dataflow graph will be active at a time.

For a J-Machine implementation of static dataflow, a token on its way to a commutative dyadic node, such as *plus*, can be represented by two words:

1. A header with the address of the destination instruction.
2. The data value.

When a token is sent to a dataflow node, this two-word message is sent to the processor on which it should run. The code for a *plus* node, written symbolically, is:

```
[Initialization code]
R0 <- CFUT:0
R1 <- MSG.VALUE + R0
[Code to send result to destination]
```

The first time this code fragment runs, the first line loads a cfuture into *R0*, signifying that the needed data is not present. When the second line tries adding the cfuture in *R0* to the new argument, a *cfuture fault* occurs because no arithmetic operations can be performed on cfutures. The cfuture fault handler, which is not built into the hardware, encodes the arriving token's value into the location of the first instruction and then suspends.[1] After this happens, the code looks like:

```
[Initialization code]
R0 <- [value of first token]
R1 <- MSG.VALUE + R0
[Code to send result and clean up]
```

When this is executed, the value of the first argument will be loaded into *R0*, and the new argument will be added to it by the second instruction. The sum is then sent elsewhere to be processed by one or more other nodes.

While the above example only works for commutative operations, it is simple to extend the strategy to non-commutative operations, such as subtraction. This can be done implicitly by having different entry points for the left and right arguments, reflected in their destination addresses, or explicitly by having code at the beginning of each node to check which token has arrived and to branch accordingly. Dally [1] goes into more detail about implementing static dataflow on the J-Machine.

2.3 Dynamic Implementation

Because static dataflow is too restrictive for most purposes, we extended the system to support dynamic dataflow, where multiple invocations of a dataflow graph can be active at once. To implement dynamic dataflow, we added an additional word to each token to hold the context pointer. Every instantiation of a dataflow graph, corresponding to a procedure application, for example, has its own context. In order to allow the code corresponding to a node to be reentrant (by keeping tokens from different invocations separate), unmatched values are not stored within the code but in a separate area of memory where the locations are determined as a function not only of the destination address but also of the context. This way, two left tokens waiting for their partners at the same node will be stored in different locations.

This process is illustrated in Figure 1. In Subfigure A, a token with the value 3 and the context *C1* arrives, causing the appropriate location, a function of *C1* and

[1]This approach would be less efficient if the encoding of the instruction to load a constant into *R0* on the J-Machine were not so simple. To load the constant *X* into *R0*, one simply places *X* in the instruction stream. When it is encountered, because it is not tagged as an instruction, the decoder knows to interpret it as a constant to load into *R0*. Thus, creating the instruction to load *X* into *R0* is trivial.

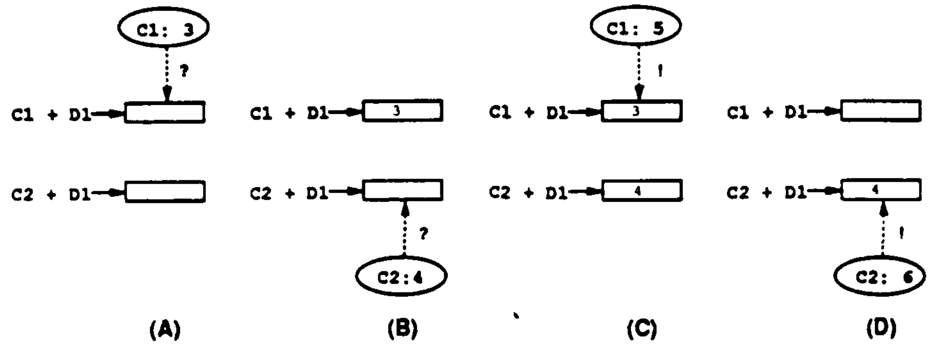

Figure 1: Snapshots for Dynamic Dataflow ETS

the destination address *D1*, to be checked for its partner. Because the location is empty, a cfuture fault occurs, and the value 3 is stored into the location. In Subfigure B, a token with the value 4 and the context *C2* arrives. Because tokens with different contexts do not interact, this token checks a different location for its partner. Finding that location empty, it stores its value there. In Subfigure C, a token with the value 5 and the context *C1* arrives. When this token checks the appropriate location, it discovers its partner, and the operation can be performed on this stored value and the newly-arrived value. Similarly, in Subfigure D, the second token with context *C2* arrives, and that operation can complete.

The MDP code for a *plus* node in the dynamic system is:

```
A1 <- MSG.CONTEXT
R1 <- MSG.VALUE
R1 <- [A1] + R1     ; This line may fault
[Code to send result and clean up]
```

First, the context pointer is loaded into address register *A1*. Next, the value of the just-arrived token is loaded into general purpose register *R1*. The addition will cause a cfuture fault if the previous token has not been stored into the location pointed to by *A1*. If the first token has arrived, *A1* will point to it, and the additionn can take place.

The cfuture fault handler is simply:

```
[A1] <- R1
suspend
```

This stores the value of the newly-arrived token into the location pointed to by the context pointer and then suspends. Note that there is no unnecessary overhead in

```
fact n =
  if n <= 1 then
    n
  else
    n * fact (n-1);
```

Figure 2: Id Code for Factorial

the code for the node or for the fault handler. Each only contains instructions to do essential loads, stores, and ALU operations.

The recursive factorial program shown in Figure 2 takes 431 ticks to compute 4!, compared to the 315 ticks the same algorithm takes on the J-Machine under Concurrent Smalltalk [8, p. 110]. A tick is the time unit used by the simulator: one tick equals one instruction, even though not all instructions on the J-Machine will take the same time. Spertus [16] describes more details of the dynamic ETS system on the J-Machine, such as the calling convention used and the mapping of instruction nodes to storage slots.

3 A Hybrid Approach

An obvious shortcoming of the ETS system is that every dataflow instruction is scheduled separately. After even the most trivial operation, such as a single addition, tokens must be built, sent, and matched. In contrast, on a hybrid system, the compiler groups *threads* of instructions into *scheduling quanta* (SQs). While this lessens the amount of run-time parallelism available, it minimizes scheduling overhead. Additionally, even dataflow computers do not attempt to exploit the maximum possible parallelism. For example, on Monsoon, a specific invocation of a procedure is generally not divided among processors but takes place on a single one. Instead, the parallelism comes from pipelining and from running iterations of a loop concurrently on separate processors [13]. A major benefit of a hybrid architecture over a pure von Neumann architecture is that the hybrid shares the latency toleration found in dataflow machines: When one branch of execution is waiting for a result from elsewhere, other instructions whose data dependencies are satisfied will be executed. A fuller justification of hybrid architectures can be found in [10, Chapters 1–2].

We did not devise our own methods to partition instructions into threads but relied on Traub's methods [21]. We built our compiler on top of Iannucci's hybrid Id compiler and, in many cases, imitated mechanisms from his hybrid architecture [10].

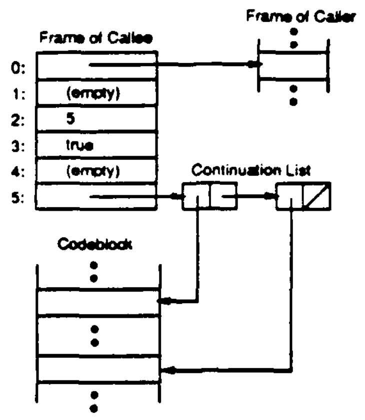

Figure 3: Run-Time Data Structures. The data for slot 5 has not arrived. The presence of a continuation list indicates that instructions in the codeblock have tried to access slot 5. When the data arrives, the SQs indicated by the continuations will be restarted.

3.1 Run-Time Structures

A procedure is divided into many SQs that together constitute a *codeblock*. When a codeblock is invoked, a contiguous region of memory called a *frame* is allocated for its arguments and scratch variables. The frame is given a unique global address, built by combining its processor number with its local address. Its first slot is initialized to point to the caller's frame. Because frames provide each invocation of a procedure with its own data area, the same procedure can be executed multiple times on one processor, with execution of the invocations interleaved. After a codeblock starts executing, it will probably fault on a slot in its frame; that is, it will look for a value in a specific slot of the frame in which no data is present, and execution of the codeblock will thus be unable to continue. In this case, a *continuation* is created encoding the address where execution should restart when the data arrives, and this continuation is stored into the empty slot. When the data arrives, it will be written into the slot and the continuation will be re-enabled, i.e. placed in the message queue for subsequent execution. When all of the SQs in a codeblock have successfully completed and any return values have been sent to the caller, the frame can be freed. These structures are shown in Figure 3 and are described in greater detail in [17]. Our compiler took hybrid code produced by Iannucci's compiler [10] as input, outputting MDP code.

```
def unsched x =
  { p = x > 0;
    a = if p then bb else 3;
    b = if p then 4 else aa;
    aa = a + 5;
    bb = b + 6;
    c = a + b;
  in
    c }
```

Figure 4: A Statically Unschedulable Codeblock. It is impossible to determine the order in which a, b, aa, and bb can be computed without knowing whether $x > 0$.

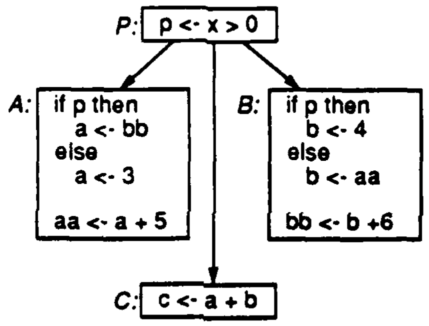

Figure 5: Scheduling Quanta for Statically Unschedulable Codeblock

3.2 Execution Within a Codeblock

To understand the execution of a codeblock, consider the procedure in Figure 4 and originally from [19, p. 2]. The order in which the statements are executed depends on whether the argument x is positive. The orders of evaluation are:

- If $x > 0, b \rightarrow bb \rightarrow a \rightarrow aa \rightarrow c$
- If $x \leq 0, a \rightarrow aa \rightarrow b \rightarrow bb \rightarrow c$

Observe that in both cases, b is evaluated before bb, a before aa, and c is last. These static dependencies allow the partitioning of the code into four scheduling quanta, as shown in Figure 5.

The order in which the SQs are executed when $x > 0$ is shown in Figure 6. P is the first SQ to run, forking A, B, and C, in that order, then suspending.

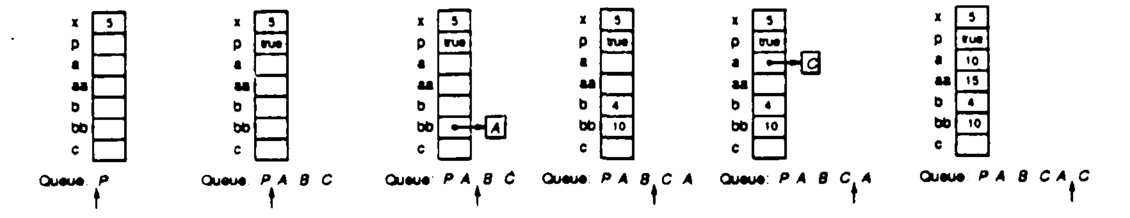

Figure 6: Snapshots of Message Queue and Frame Memory for Program in Figure 2

A begins, then suspends, because *bb* is needed but not available. B, next in the queue, begins and runs to completion. When it stores *bb*, it sees that A is waiting on the value and sends a message to restart A. C then begins executing and faults on *a*, suspending. The second request to execute A is now at the head of the message queue and completes, sending a request to restart C. C runs, performing the addition and whatever else follows, such as returning the resulting value.[2]

3.3 Loops

As encouraged by the semantics of Id, up to K different iterations of a given loop can be active at a time, where K is the dynamic loop-unfolding constant. Because iterations of an outer loop nominally run on the same processor, individual instructions are not executed concurrently. Instead, the parallelism comes from two sources: First, when a calculation within one iteration is waiting for data, such as the result of a procedure call to another processor, instructions from other active iterations may be executed, subject to data dependencies. Second, if there are subroutine calls or inner loops within the loop, these may be spawned onto other processors and run in parallel, as will be described in more detail below. Because up to K iterations of a loop may be active at once, there must be K iteration areas allocated to store the intermediate values. Thus, frames for codeblocks with loops are larger than frames for non-loop codeblocks. The methods for ensuring correctness when parallelizing loops are inherited from Iannucci's compiler, and the interested reader is referred to [10, Section 4.3.5].

Figure 7 shows a simple program to sum the results of a function applied to the first n positive integers. The keyword *next* allows the circulating variable *total* to be referred to in a functional way. The loop variable *count* is also a circulating

[2]The reader may have observed that the sample procedure could be totally statically scheduled by observing that it returns 14 if $x > 0$ and 11 otherwise. The example is still relevant, because procedures exist for which no such reduction is possible, for example, if the bindings for a and b were changed to `a = f x bb` and `b = g x aa`, where f and g were passed in as parameters [19, p. 2]. The simpler program is used purely for ease of exposition.

```
def add_em_up f n =
  { total = 0
  in
    { for count <- 1 to n do
        next total = total + f count
      finally total }}
```

Figure 7: An Id Program to Sum a Function Applied to the First n Integers. Loop keywords are "for", "next", and "finally".[4]

variable, i.e. it is read and written by iterations of the loop. Iteration areas with space for these variables, plus temporaries to handle the procedure call, are allocated within the frame when the procedure is invoked. The initial values of *count* and *total* are set in the first iteration area, which gets marked as enabled, meaning it may run. For each enabled iteration, the following steps occur:

1. Compare *count* to n.
2. If *count* $\leq n$ then
   (a) Write *count*+1 into the appropriate slot of the next iteration area and mark it as enabled.
   (b) Spawn the procedure call, *f count*.
   (c) Add the result of the previous step to *total*, writing the sum into the *total* slot in the next iteration area.
   (d) Because all reads of the incoming circulating variables have been performed, set a flag in the previous iteration area to indicate that the iteration area may be reused.[5]
3. If *count* $> n$ then write the current value of *total* to a frame slot outside the iteration areas.

After the final result has been written to the outside frame slot designated for the *finally* value, the appropriate SQ will run, sending the value to the caller and freeing the codeblock frame. Figure 8 illustrates the process by showing the contents of the first three iteration areas as the loop begins. Enabled iterations

[4]A more efficient way to perform the same function would be to use a tree, but this program is better for illustrating relevant parts of our system.

[5]For the full details on how iteration areas are recycled, i.e. when the dynamic number of iterations is greater than the number of iteration areas K, see [17, pp. 20–25].

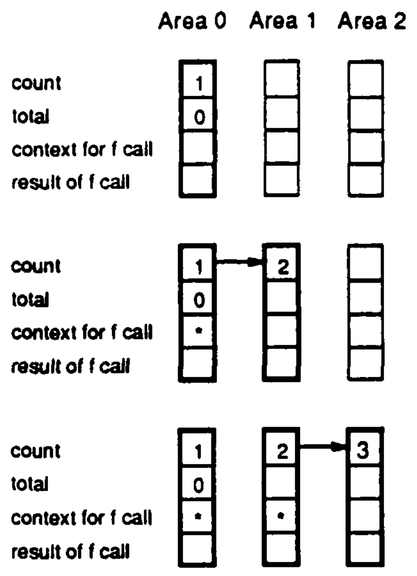

Figure 8: Snapshots for Loop Example

are denoted with bold borders. Because up to K invocations of f will run at once, this method can give substantial speed-up if f is slow.

The reader will observe that this scheme does not address nested loops. These are lifted out of procedures at compile-time and form new codeblocks that will be called by the original procedure. Thus inner loops can run in parallel on separate processors.

3.4 Performance

Because we used a simulator of the J-Machine, we only were able to run small programs and to simulate a small number of processors (up to 32). Larger-scale experiments await the construction of the J-Machine. The two programs we analyzed in greatest depth were a recursive factorial procedure and a simple procedure that used the loop constructs.

3.4.1 Factorial

One test program was the recursive factorial program shown in Figure 2. In our preliminary system, load-balancing is not performed by the compiler. We modified the MDP code to spawn the factorial call with argument n to processor n to distribute processor usage. The total code length is 180 MDP instructions, not counting code in library routines [17, Section 4.1].

Handler Name	*# Calls*	*Ticks/Call*	*Total Ticks*
Lookup	25	5 or $6 + 6w$	212
Cfut	21	18	378
Move-Remote	16	$13 + 7w$	264
Continue-Test	14	7 or 20	189
Get-Context	3	24	72
Allocate	3	12	36
Total	136	n/a	1061

Table 1: System Calls for `(fact 4)`

When 4! is computed on the MDP simulator, it takes 1263 ticks for the result to be written to the original calling frame. With four processors enabled, utilization is 37% — i.e. on average, a processor does useful work a little over a third of the time. The hybrid system does not use the operating system written to support object-oriented programming on the J-Machine [8], so the library routines and fault handlers constitute the entire operating system. The functions of these routines are:

- Lookup — Check if a frame slot holds a disabled continuation before writing to it. If a continuation is present, enable it to signify that the data has arrived.
- Cfut — Store the current continuation in the slot on which the cfuture fault occurred, then suspend.
- Move-Remote — Send a value into a frame slot of a parent or child procedure, usually on another processor.
- Continue-Test — Test whether a continuation will be able to run, before enabling it.
- Get-Context — Allocate and initialize a frame locally, send the address to the calling procedure, and enable the first SQ in the codeblock.
- Allocate — A low-level memory allocation routine used by get-context and cfut.

Fault and library usage are shown in Table 1. As the totals show, 84% of the time is spent in the operating system. The routine that consumes the most time is the cfuture fault handler. It is called 21 times, and each time takes 18 ticks. It

Argument	*Ticks/Call*	*Ticks/Skewed Calls*		*Ticks/Nonskewed Calls*	
		1st Result	2nd Result	1st Result	2nd Result
4	1263	1864	2163	1992	2271
8	2691	4204	4590	4332	4611
12	4119	6544	6846	6672	6951

Table 2: Latencies for Factorial. This table compares the number of ticks required to compute one and two calls of factorial. For each case, the number of processors used is the same as the argument. The first data column shows how long it takes for one call executing alone to complete. The second set of columns shows how long it takes to complete two factorial calls initiated at the same time, skewed among the enabled processors. The last set of columns shows the completion times when the two calls are not skewed among the processors.

takes so long because it must allocate space to store a continuation and fill in the necessary data.

As described in Section 2.3, the ETS implementation takes 431 ticks, compared to the hybrid system's 1263. The cost of using library routines to simulate Iannucci's architecture was very high. A more efficient approach would involve less run-time interpretation.

Another reason why computations take a relatively long time to complete is that many design decisions favored throughput over latency. This is due to the decision to break apart any transaction of unbounded latency, which increased the latency of tasks but improved throughput. Table 2 shows that computing two invocations of factorial concurrently on the J-Machine takes significantly less than twice as long as computing a single call. This is true for two reasons:

1. Each task suspends itself when it is waiting for a result from another processor, allowing the processor to be used by another task.

2. The factorial calls can be skewed among the processors, i.e. by mapping the calls to processors in such a way that interference between the two calls is minimized.

Table 2 isolates these factors by including results for when the procedure calls are skewed and when they are not. Even when two factorial calls are executed on the same processors, in the same order, throughput is increased over the single call case. This is because subtasks of the second factorial call can run when no work can be done on a given processor toward the first factorial call.

```
def loop n =
  { sum = 0
   in
  { for i <- 1 to n do
      sum_increment = sum + i;
      next sum = sum_increment
    finally sum }};
```

Figure 9: Id Code for Loop Example[6]

3.4.2 Loop Parallelization

The loop program we used is shown in Figure 9. The program returns the sum of the first n integers.

There is a straightforward method for choosing processors for procedure calls made from loop bodies so that they do not conflict with each other: If there are K iteration areas, each can be assigned unique processors to send subcalls to; for example, iteration area i can spawn its subcalls to processors i, $i + K$, etc.

This program is a useful benchmark in that it shows the overhead to set up iteration areas and to launch iterations of a loop in parallel. The number of ticks, as a function of K (the number of iterations to unroll) and n (the argument) was $50+5*K+135*n$. The three addends of the formula can be interpreted as follows:

1. The constant term, 50, indicates that the additional cost for a procedure to use loop parallelization is low. There is thus little inhibition against parallelizing loops.

2. The $5*K$ term is a pleasant surprise: Once the base cost for loop parallelization has been paid, it only costs 5 ticks to add and support each iteration area. This makes it reasonable to unroll many iterations of a loop.

3. The $135*n$ term shows the overhead involved in managing the flags and circulating variables in each iteration of the loop. If each iteration of the loop spawns a long subroutine, as in the example in Figure 7, the only additional code that will run on the home processor is that to spawn a procedure call. This means that each iteration of the loop will use fewer than 200 ticks on its home processor, regardless of how big a computation it performs. As described above, it is trivial to distribute its procedure calls so that they do not interfere with those of other iterations.

[6]The body of the loop could have been written more succinctly as `next sum = sum + i`, but compiler compatibility issues dictated the use of the more verbose form.

Number of input tokens waiting in structure
Slot for first token
Slot for second token
⋮
Slot for last token

Figure 10: New Frame Format

4 Conclusion

4.1 Future Research

Our implementation of two dataflow systems demonstrates that the J-Machine has the necessary mechanisms to execute dataflow programs. However, certain aspects of each system can be improved. The hybrid system did too much run-time imitation of Iannucci's architecture, which was especially costly because his compiler was optimized for operations cheap on his architecture but expensive on ours. The ETS system gave better results, but a large percentage of its execution time went to overhead which could be avoided if nodes of the dataflow graphs were combined.

We now plan to build a system that exploits the strengths of each method. Specifically, we want to retain the scheduling quanta and loop parallelization of the hybrid system and the low overhead of ETS. We believe this can be done by basing our new system directly on the Id compiler used by the MIT Computation Structures Group [18] instead of using Iannucci's machine language as an intermediate step, only making use of his routines to parallelize loops and Traub's routines to partition a dataflow graph into scheduling quanta [21]. With Traub's SQs, we expect the new system to be several times faster than our ETS system.

We now plan to build a hybrid ETS system based on the observation that each SQ has a fixed number of tokens used as inputs, where the number of input tokens is known at compile-time. Instead of performing a single operation, as the ETS system does, when the argument tokens arrive, this system would execute an entire SQ after all the input tokens arrive. Because SQs created from Id programs typically consist of a relatively small number of instructions, this will not significantly lessen the parallelism, and the runtime overhead will be substantially reduced. Figure 10 shows what the data structure would look like. Its first slot would hold a count of the number of arguments to the SQ that have arrived, and subsequent slots would hold these arguments. When an argument arrives, if it is the last one, the SQ will be executed; otherwise, the token will be stored and the count incremented. This system would be more efficient than the ETS system because of the less fine-grained scheduling: Instead of scheduling one machine instruction at a time, it schedules several. Its primary advantage over the hybrid system is that the expensive lookup system call and cfuture fault handler used to imitate Iannucci's architecture (Section 3.4.1) would be entirely unnecessary.

4.2 Implications for the J-Machine

Our results with dataflow on the J-Machine demonstrate the ability of its mechanisms to support fine-grained asynchronous programming models. For example, although loop support was not specifically built into the J-Machine, the existing mechanisms allowed the implementation of Iannucci's style of parallelizing loops. This research has also identified areas where the mechanisms could be improved to reduce overhead. The high costs of the cfuture fault handler and the lookup system call (Table 1) of the hybrid system suggest the need for more hardware support for synchronization, such as hardware mechanisms to suspend a task when a cfuture is encounted and to automatically restart a continuation when the data arrives. This would greatly reduce overhead for the dataflow model as well as for shared-memory and object-oriented programming models. Even though the proposed system could avoid those relatively slow routines by using the ETS system's less expensive routines, the hardware mechanisms would be useful for providing I-structure memory, an important item in dataflow systems, which now must be simulated in software.

Acknowledgments

We would like to thank the members of the MIT Concurrent VLSI Architecture group for their help throughout the project, including Lisa Sardegna who helped us prepare this paper. Additionally, Arvind, Jamey Hicks, Robert Iannucci, and Jonathan Young of the MIT Computation Structures Group were very generous with their expertise and equipment. We are grateful to John Zahorjan and Cecelia Buchanan of the University of Washington for providing one of the authors with a computer account to work on this paper when away from MIT. We thank André Dehon and Nate Osgood of MIT, Jeri Steele of Texas Instruments, and Helen Takacs of Mississippi State University for their many helpful comments on earlier versions of this paper.

References

[1] Dally, William J. Dataflow on the J-Machine. An unnummbered MIT Concurrent VLSI Architecture memo, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[2] Dally, William J., et al. The J-Machine: A Fine-Grain Concurrent Computer. *Information Processing 89, Proceedings of the IFIP Congress*, 1989.

[3] Dally, William J., et al. Message-Driven Processor Architecture. MIT Artificial Intelligence Laboratory Memo 1069, Cambridge, MA, 1988.

[4] Dally, William J. and Wills, D. Scott. "Universal Mechanisms for Concurrency" in Goos, G. and Hartmanis, J., eds. *Proceedings of PARLE-89*, Springer-Verlag, 1989.

[5] Davis, A. L. "A Data Flow Evaluation System Based on the Concept of Recursive Locality" in *The Proceedings of National Computer Conference*, 1979, pages 1079–1086.

[6] Gaudiot, Jean-Luc and Lubomir Bic. Data-Flow: A Status Report. *Computer Architecture News*, December 1989, pages 111-118.

[7] Gurd, John and Watson, Ian. "Data Driven System for High Speed Parallel Computing — Part 2: Hardware Design" in *Computer Design*, July, 1980.

[8] Horwat, Waldemar. Concurrent Smalltalk on the Message-Driven Processor. MIT Artificial Intelligence Laboratory Technical Report 1080, Cambridge, MA, 1990. (Master's Thesis, Department of EECS, MIT.)

[9] Horwat, Waldemar and Brian Totty. Message-Driven Processor Simulator. MIT Concurrent VLSI Architecture Memo 5, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1987.

[10] Iannucci, Robert Alan, A Dataflow / von Neumann Hybrid Architecture. Technical Report MIT/LCS/TR-418, MIT Laboratory for Computer Science, Cambridge, MA, 1988. (PhD Thesis, Department of EECS, MIT.)

[11] Nikhil, Rishiyur S., ID Version 88.1 Reference Manual. Technical Report Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[12] Papadopoulos, Gregory Michael. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, MIT Laboratory for Computer Science, Cambridge, MA, 1989. (PhD Thesis, Department of EECS, MIT.)

[13] Papadopoulos, Gregory M., and David E. Culler, Monsoon: An Explicit Token Store Architecture. In *Proceedings of the* 17th *International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[14] Sakai, Shuichi; Yamaguchi, Yoshinori; Hiraki, Kei; Kodama, Yuetsu; and Yuba, Toshitsugu. An Architecture of a Dataflow Single Chip Processor. *Proceedings of the* 16th *Annual International Symposium on Computer Architecture*, Jerusalem, Israel, 1989, pages 46–53.

[15] Shimada, Toshio; Hiraki, Kei; Nishida, Kenji; and Sekiguchi, Satosi. Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations. In *Proceedings of the* 13th *International Symposium on Computer Architecture*, 1986, pages 226–234.

[16] Spertus, Ellen. Preliminary Dataflow on the MDP. MIT Concurrent VLSI Architecture Memo 21, MIT Artificial Intelligence Laborary, Cambridge, MA, 1989.

[17] Spertus, Ellen. Dataflow Computation for the J-Machine. MIT Artificial Intelligence Laboratory Technical Report 1233, Cambridge, MA, 1990. (Bachelor's Thesis, Department of EECS, MIT.)

[18] Traub, Kenneth R. A Compiler for the MIT Tagged-token Dataflow Architecture. Technical Report MIT/LCS/TR-370, MIT Laboratory for Computer Science, Cambridge, MA, 1986. (Master's Thesis, Department of EECS, MIT.)

[19] Traub, Kenneth R., Compilation as Partitioning: A New Approach to Compiling Non-Strict Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[20] Traub, Kenneth R. A Dataflow Compiler Substrate. Computation Structures Group Memo 261, MIT Laboratory for Computer Science, Cambridge, MA, 1986.

[21] Traub, Kenneth R., Sequential Implementation of Lenient Programming Languages. Technical Report MIT/LCS/TR-417. MIT Laboratory for Computer Science, Cambridge, MA, September 1988. (PhD Thesis, Department of EECS, MIT.)